AVANCE:
An Object Management System

Anders Björnerstedt
Stefan Britts

SYSLAB †
Department of Computer and Systems Sciences
University of Stockholm
S-106 91 Stockholm
SWEDEN

Abstract

AVANCE[1] is an integrated application development and run-time system. It provides facilities for programming with shared and persistent objects, transactions and processes. The architecture is designed with decentralization in mind by having a large object identifier space and a remote procedure call interface to objects. Emphasis in this paper is on the programming language PAL and its relation with the underlying virtual machine.

1. Introduction

Over the last years there has been an increasing interest in object oriented programming languages (OOPLs) and object oriented systems. OOPLs range from languages which simply encourage or at least permit an object oriented programming style [Birtw73, Cox86, Strou86], to languages which permeate the object oriented "paradigm" in every part of the language [Goldb83]. Recently attempts have also been made to integrate such languages with facilities for making objects persistent and shared, that is, to provide some kind of data base for storing objects [IEEE85, IEEE86]. This has been accomplished either by making extensions to existing languages [Maier86], by extending the data model of an existing DBMS [Stone86], or by designing entirely new languages and systems [Niers87]. Idealy, the language and the data model should be seamlessly integrated. If the language and the data model can become one and the same then the programmer is relieved of problems such as converting between representations suitable for the database and the application.[2]

The interest in object-orientation as a basis for designing and implementing information systems is based on the recognition of advantages, both on the level of modeling and design, and on the level of implementation with the support possible for software construction. Object-oriented architectures can model reality closely by allowing an extensible data model. For the purposes of software construction and programming support, object-oriented architectures provide modularity, flexibility, support for change and reuse of software, and for generic programming.

Object-oriented architectures are also promising for constructing decentralized and loosely coupled (or open) systems [Gray86, Hewit84]. Various forms of late binding may be incorporated at strategic points in the system to allow for flexibility and openness with respect to time, structure, location etc, without compromising local consistency.

† This work is supported by the National Swedish Board for Technical Development (STU).

[1] AVANCE was previously named OPAL. It is under development at the University of Stockholm and SISU (the Swedish Institute for Systems Development).

[2] This is sometimes called providing a *single level store*.

One can generalize the concept of an object-oriented database to what in this paper will be called an *Object Management System* (OMS). An OMS is analogous to an operating system in that it provides a complete computational platform for both development and execution of applications. An OMS provides a more homogeneous "world view" than an object-oriented database. Not only are both objecttypes and instances regarded as objects, but compilers, processes, devices, ..., everything available is regarded as an object. No distinction is made between a database acting as a repository and external applications for providing functions to users. In an OMS it should ordinarily not be necessary to go outside the system for some particular functionality, like application design. As an example, standard Smalltalk-80 could be called an OMS except that multiprogramming (true parallelism), sharing and atomic operations (transactions) are not available. GemStone[3] does provide database facilities in a Smalltalk environment, but has two object models (the Smalltalk and GemStone models) which are not completely unified [Purdy87].

At this point we want to make explicit two points of possible misunderstanding regarding the concept of OMS as we use it here. The first concerns the reason why we are advocating the OMS architecture. It has been pointed out that building a database on top of a general operating system creates problems of efficiency [Stone81, Gray78]. Although this may be a reason for integrating operating systems with databases, efficiency is not our primary concern here. Our primary concerns are homogeneity, extensibility and reliability.

The second point concerns homogeneity. Although the goal of an OMS should be to provide all the functionality users need within the system, it does not have to be closed. The homogeneity provided by an OMS is something to be seen as exploitable, not as something constricting. An OMS should then provide "windows" to the outside world, which may be used for reasons of efficiency and practicality. If such "windows" to other worlds are to be provided then these should also be provided in terms of objects. Object-oriented databases do have the advantage that they will probably be more efficient when integrated with existing systems. This is because object-oriented databases are often designed as servers [Horni87, Purdy87], while an OMS is primarily designed to serve itself. Because an OMS is more than a database, it should be possible, given the appropriate primitive types, to design the appropriate objecttypes making the OMS behave as a server. However, since the OMS is not optimized for serving external applications it will probably not be able to compete as a general server with object-oriented databases, or DBMS tool-boxes such as EXODUS [Carey86].

Homogeneity encourages the reuse of software and the incremental development of applications. It also makes it simpler to keep the entire system consistent. However, it may also create problems. One problem that an OMS must face, that object-oriented databases do not have, is the interference between transaction management and user interface management. If there is a strict borderline between the database and applications, then transaction management naturally is the responsibility of the former, while user interface management is the responsibility of the latter. In an OMS no such strict borderline exists and as a consequence the relationship of transaction- and user interface management needs to be reexamined.

AVANCE [Ahlse84, Ahlse85], is a research prototype OMS. It is a fairly large and complex system. This paper focuses on the language PAL, and the parts of AVANCE of concern to the programmer. AVANCE supports, among other things, sharing of persistent objects, nested transactions, object version management, decentralization of both data and control, a strongly typed compiled programming language, and a weakly typed interpreted command language. Aspects of AVANCE which will *not* be discussed in this paper are the user interface, authorization, exception handling, views, and triggers.

Decentralization is an important aspect of AVANCE and this has had effects on the system design. Semantic issues of type equivalence, type changes, and naming have been addressed. Although decentralization has not been fully implemented yet in AVANCE, much effort has been made in the design to make this step as painless as possible. AVANCE is a running system. The implementation effort so far has focused on architectural issues and not on the user interface, the type library, or development tools.

Of the other systems we have looked at the one perhaps most similar to AVANCE is VBASE [Andre87]. VBASE emphasizes an integrated language and database environment, with database facilities more powerful than what is currently available in AVANCE. For example, AVANCE does not have system support for bi-directional relationships (one-to-one, one-to-many and many-to-many). On the other hand AVANCE will support decentralization which seems to be missing from the VBASE architecture. In this respect AVANCE is more like the Emerald system [Black86].

The paper is structured as follows. First an overview of AVANCE, then the PAL programming language, and

[3] GemStone is a registered trademark of Servio Logic Development Corporation.

finally, the AVANCE virtual machine.

2. Overview of AVANCE

The architecture is designed to provide three levels of abstraction to implementors. Figure 1 illustrates the static system software components of AVANCE.

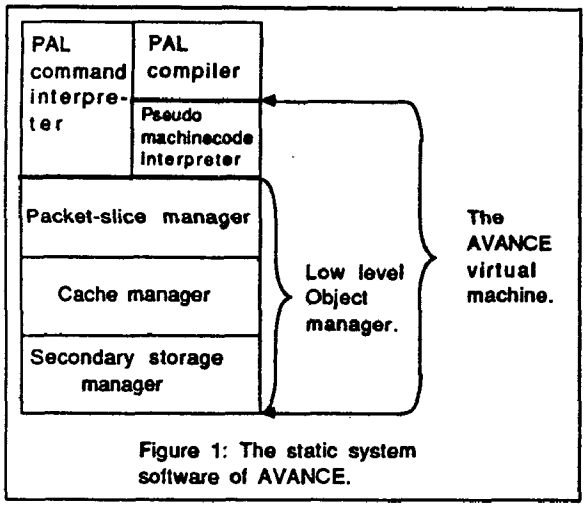

Figure 1: The static system software of AVANCE.

The low level object manager provides functions for identifying and manipulating a data abstraction called the *packet-slice*. This concept will be explained further in section 4. It provides an interface suitable for building interpreters.

The second level of abstraction is the virtual machine interface provided by the pseudo-machine code (p-code) interpreter. It has similarities with the Smalltalk-80 virtual machine [Goldb83]. The interface provided by the p-code interpreter is essentially a stack machine which understands a set of primitive datatypes and an instruction repertoire to operate on these. The interfaces provided by the object manager and the p-code interpreter are not intended for human interaction.

The third level of abstraction is the high level language PAL. A compiler translates PAL into the pseudo-machine code of the virtual machine. It is at this level that the full power and homogeneity of an OMS is available. Most users / programmers would only use this level of abstraction.

The low level object manager is relatively independent of the p-code interpreter, and the p-code interpreter relatively independent of the PAL compiler. Thus it is possible to have several alternative interpreters on top of the object manager, and several different compilers on top of an interpreter. This makes it possible to have several different programming languages and virtual machines, or different versions of them simultaneously running in the system. Presently, besides the p-code interpreter and PAL compiler already mentioned, there is only a simple command language interpreter which directly executes a subset of PAL.

2.1. Object Granularity

The world of objects in PAL is partitioned into two disjoint sets: *packets* and *datatype values*. Both packets and datatype values are instances of abstract datatypes. By this is meant that they have an internal hidden representation and the only way to operate on a packet or a datatype value is by using an operation defined for the type.

2.1.1. Packets

Packets are associated with persistence, independent existence, synchronized sharing, resiliency [Svobo84], atomicity [Lisko83], and system version control. Packets are identified by non-reusable surrogates allocated when the packet is created. The identifier space is very large in order to allow global identification in a decentralized / distributed system. Packet identifiers are only used as components of packet references, and are not directly available for manipulation outside the object manager. Packets may be aggregated by the use of packet references into any directed graph structure. Packets may be shared between AVANCE processes.

A Packet is roughly comparable to a *monitor* [Hoare74], in that it synchronizes processes which invoke operations on it. A packet operation invocation is a *remote procedure call*. In other words, arguments and return values are passed by value, the invoker is blocked until the invoked operation returns control, and the details of communication protocol and error detection are hidden from the programmer. Packets resemble Guardians in Argus [Lisko83] in that they are persistent objects with remote procedure call handlers. They differ from Guardians in that they are generally not active by themselves (they do not have their own thread of control).

2.1.2. Datatype Values

Datatype values do not have independent existence. For a datatype value to be persistent it has to be assigned to an instance variable of a packet. Datatype values never overlap, that is, every instance variable contains a "private" value, and assignment of values between variables will entail a copying of values. This makes garbage collection relatively simple[4], but reduces the efficiency of the system. The only way for two or more objects to share an object is by using references to a common packet.

[4] Garbage collection of datatype values is handled automatically by the virtual machine. Garbage collection of packets has not been implemented yet, but the intention is to do it by purging unreachable packets on a per-node basis (see section 2.2).

Datatype values have significantly less overhead than packets. The operations of datatypes can be implemented rather efficiently, compared with the operations of packettypes, since no overhead for persistency, synchronization, version control etc. is needed. Furthermore, datatype values are not assigned system unique identifiers (like packets), since they are not persistent, independently existing objects. This also reduces space and time overhead. For example, the object space is not "polluted" with huge amounts of small or temporary objects. Unless a datatype value has been assigned to an instance variable of a packet or given as a return parameter, it will be garbage collected when the packet operation which created it returns. Depending on these differences in properties, the designer can decide whether to implement a type as a packettype or datatype.

The PAL programmer defines the internal representation of a type (both packet- and datatypes) in terms of datatypes, i.e., by declaring instance variables on datatypes. Packets can not be directly incorporated in the internal representation of a type, instead the reference datatype is used. The packet reference datatype is the "bridge" between the datatype type system and the packettype type system. A programmer may create a datatype which in its representation has packet references, thus the distinction between packettypes and datatypes as the former being persistent etc. and the latter not must be taken in a shallow sense[5].

The p-code interpreter implements a number of primitive datatypes. Most of these directly correspond to datatypes at the PAL level and are available to the PAL programmer, while some of them are only used by the compiler for its own purposes.

2.2. The AVANCE Logical Network

The large scale architecture of an AVANCE system consists of a logical network of nodes[6] (see figure 2). Each node provides a centralized information processing environment, suitable for an "organizational unit" requiring authority over processing and data. Protection of data is handled within a node. Each node resides on a physical machine *host*. Each node may become unavailable for communication with other nodes in the system. This may be because the administrator of the node has decided to close it, or due to a crash of the node or host, in which case the node closes without warning.

[5] The word *shallow* should be taken in the sense of *shallowCopy* or *deepCopy* in the *Object* instance protocol of Smalltalk-80.

[6] The implementation of AVANCE has currently only progressed far enough to run single nodes in isolation.

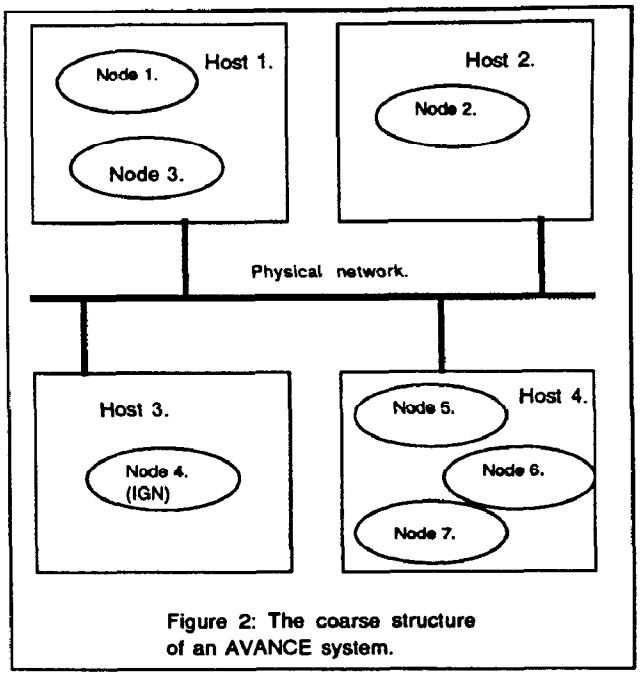

Figure 2: The coarse structure of an AVANCE system.

Nodes are thus relatively autonomous and what unites a set of AVANCE nodes into a system is the adoption of:

- a common identification scheme for packets,
- a common representation form for all packets and datatype values,
- a communications protocol between nodes, and
- a common set of packettypes and datatypes.

These four points will be detailed in section 4.2. We require that all nodes run the same version of object manager software.[7] The reason for this is that the object manager maintains a lowest level of consistency in the system. Identification scheme, representation form, and communication protocol are all defined at this level. In practice, nodes will also run the same interpreter(s). If nodes are to have any meaningful communication they will at least have to agree on a set of common primitive datatypes. But, as mentioned earlier, they could run different versions of the same interpreter, or different interpreters with some primitive datatypes in common.

3. The PAL Programming Language

PAL is a high-level, block structured language primarily inspired by Simula [Birtw73], Smalltalk-80 [Goldb83], and CLU [Lisko81]. It contains facilities for both defining and manipulating objects. In addition to ordinary programming language constructs, func-

[7] AVANCE is implemented on Sun UNIX using the C programming language. For the forseable future we require all hosts to be Unix machines.

tionality for persistency, version management, concurrency and decentralization are supported. Facilities to support exception handling and triggers are currently under development [Ahlse87].

All facilities are completely integrated within PAL. Hence, there is no difference between, on the one hand defining and manipulating "program objects" and on the other hand defining and manipulating "database objects". PAL supports typing, property inheritance, data abstraction, encapsulation, instantiation and both dynamic and static binding within a homogeneous environment. Not only "data", but also "meta-data", are represented as objects.

In spite of its strength, PAL is a fairly small language. This is thanks to the extensible data model and the uniformity possible with object-orientation. A limited number of syntactical constructs and pre-defined objecttypes cover most basic needs. What is not covered can usually be added by defining new objecttypes. Logically, there is no difference between pre-defined and user-defined objecttypes.

3.1. Typing and Instantiation

In order to illustrate the capabilities of PAL as a design and implementation tool a simple electronic mail application will be used as an example.

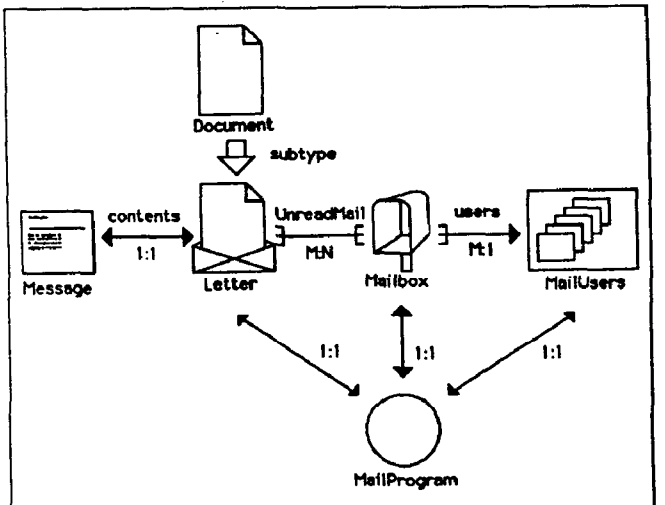

Figure 3: Conceptual model of e-mail application

A user gets access to the mail system by executing an operation *main* on an instance of the *MailProgram*.[8] In order to be able to receive letters a user must be registered in *MailUsers*. The mail user register is used by the application for looking up the recipients mailbox(es) when sending letters. A *Mailbox* is associated with each registered user. All unread letters are stored in this. *Letter* is the type of object communicated among users. Letter is a subtype to *Document*, which is assumed to be a generic "business document", generalizing properties of letters, memos, reports, etc. The letter contents type, finally, is called *Message*.

As packettypes and datatypes have somewhat different semantics one has to decide whether to define each entity type in the conceptual model as either a packettype or as a datatype. In the e-mail application *Document, Letter, MailProgram, Mailbox* and *MailUsers* are defined as packettypes. The primary reasons for this is that we want objects of these types to have an independent existence or to be shared among users. *Message* (the type of the letter contents) is defined as a datatype. This is possible since a message is always associated with a letter (which is a packettype) in a 1:1-fashion. If we wanted the ability to send the same *Message* object in more than one *Letter*, then we would also have made *Message* a packettype.

In order to have a running e-mail application the objecttypes above must be programmed, compiled, installed and instantiated. In principle, any number of instances may be created of a type. In this example, however, there must be exactly one instance of *MailUsers* since this object will act as a central register. In general, some types need not be instantiated. A type without any instance variables does not need any instance to execute. A "program" in the traditional sense (without private persistent data) would be implemented as an operation which does not access any instance variables. In this case it would not make any difference whether the type was implemented as a packettype or as a datatype. There may also be types which are not intended to be instantiated as the most specialized type of an instance. These are called *abstract types* and are used only to generalize properties of subtypes. As in Simula and C++ an operation may be declared *virtual* in order to defer the implementation to subtypes. *Document* is an example of an abstract type.

3.2. Property Inheritance

Both datatypes and packettypes are organized in property inheritance "trees". As *multiple inheritance* is supported the inheritance "tree" is rather a directed acyclic graph. In figure 4a, parts of the inheritance graph for primitive datatypes (strongly inspired by Smalltalk) is shown. Indentation means property inheritance.

[8] *MailProgram* is actually a subtype of *Application*, which is a system-defined packettype (see below). It is debatable whether *MailProgram* and *Application* really are conceptual entity types. They have been included here only for reasons of completeness.

Primitive Datatypes	System-Defined Packettypes
• Value	• Packet
• • Boolean	• • Node
• • Magnitude	• • DirectoryEntry
• • • Date	• • • Directory
• • • Number	• • • Application
• • • • Integer	• • • • TypeDef
• • • • Real	• • • • • DatatypeDef
• • Compiler	• • • • • PackettypeDef
• • Parameterized	
• • • Collection	
• • • • Sequence	
• • • • Bag	
• • • Reference	
• • • Process	
(a)	(b)

Figure 4: Some Pre-Defined Objecttypes

Since datatypes and packettypes have different semantics, a datatype cannot be a subtype of a packettype and vice versa. Therefore, a separate inheritance graph is constructed for packettypes. In figure 4b, some system-defined packettypes are shown.

All datatypes and packettypes must, directly or indirectly, be a subtype to *Value* or *Packet* respectively.[9] Hence, user defined objecttypes may not form separate inheritance graphs. If a new objecttype is not explicitly declared to be a subtype of an existing type, it is assumed to be a subtype of one of the two root-level objecttypes.

Some datatypes are *type parameterized* (unlike Smalltalk), e.g., *Collection, Sequence, Array, Set,* and *Reference.* Parameterization can be seen as a weak form of inheritance. A parameterized datatype is usually a structured type where the parameter specifies the type of its' component(s), e.g., *List(Integer), List(Char),* and *Array(List(Set(Integer))).* The *Reference* datatype (*Ref* for short) is a special case in that it is parameterized with a packettype, while most other parameterized types are parameterized with a datatype. An instance of *Ref* is a handle to a packet of the type the parameter specifies, e.g., *Ref(Letter),* and *Ref(Mailbox).*

3.3. Data Abstraction and Encapsulation

Following the data abstraction philosophy, objecttypes are defined in terms of a specification and an implementation. These parts are defined separately and may also be compiled separately. An application may be defined purely by specifications and these compiled before the corresponding implementations exist. After this several programmers may work independently on the implementations. The PAL compiler ensures that specifications are type compatible with each other and that implementations conform to their specification.

3.3.1. Specifications

An objecttype specification consists of three optional parts: a context part, a public part, and a private part. In the *context part* supertypes plus packettypes and datatypes that this particular objecttype needs to have access to are stated. Note that primitive datatypes need not be stated. The *public part* contains declarations of operations that are available to all other objecttypes, which use this type. The *private part* declares operations that are available only to subtypes of this objecttype and to itself for operating on itself and other instances of the same type. Operations may return multiple results (as in CLU).

A specification thus defines two interfaces. The interface provided by the public operations for external use, and the interface provided by both the public and the private operations for use by subtypes and in its own implementation. The private operations makes it possible to provide tools to implementors of subtypes, without making these tools visible externally. The private operations also serve another purpose. It is sometimes necessary for a type to operate on more than one instance of itself in the implementation of an operation, e.g., anInteger.add(anotherInteger). In CLU this is handled by an explicit conversion of abstract objects to their representation. In PAL the problem is solved by defining a private operation returning the internal representation. If we only had the public interface then the operation would be accessible to any other objecttype, which would violate encapsulation.[10]

The packettype specifications for *Document* and *Letter* in the e-mail example look like this:

[9] This is a constraint enforced by the PAL compiler and not inherent to the object manager.

[10] Because the private operations serve two different purposes, as an interface towards subtypes, and as an interface towards other instances of the same type, we long deliberated over whether to have three interfaces in the specification. We decided against this because we thought the price in terms of a more complicated language was not worth the added security. Both purposes of the private operations also in a sense serve implementors of "one" type.

```
PACKETTYPE Document;
 SPECIFICATION
  /* Generic "Business Document" */
 PUBLIC
  OPERATION Edit(author : Text); VIRTUAL;
  OPERATION toListOfText() : List(Text); VIRTUAL;
  OPERATION getAuthor() : Text;
 PRIVATE
  OPERATION putAuthor(name : Text);
  OPERATION putLastUpdate();
END Document;

PACKETTYPE Letter;
 SPECIFICATION
  /* The type of object communicated among users */
 CONTEXT
  SUPERTYPE Document;
  PACKETTYPE Mailbox;
  DATATYPE Message;
 PUBLIC
  OPERATION Send(mailboxes : List(Ref(Mailbox)); To:List(Text));
  OPERATION getDateSent() : Date;
END Letter;
```

The context part of *Letter* states that it is a subtype to *Document* and that it must have access to the *Mailbox* packettype and the *Message* datatype. Two public operations are declared - *Send* and *getDateSent*. No private operations are declared. *Letter* inherits the context and all public and private operations of *Document* and *Packet*. *Packet* is an implicit supertype of *Document*. PAL-defined datatypes are almost identical to packettypes syntactically.

3.3.2. Implementations

An objecttype implementation consists of two parts, a *representation* and an *operation implementation*. The representation defines instance variables used to hold the object state. The operation implementation defines operations. Below, parts of the implementation of *Letter* is shown:

```
PACKETTYPE Letter;
 IMPLEMENTATION

 ATTRIBUTE
  contents : Message;
  DateSent : Date;

 OPERATION Edit(author : Text); /* Edit letter contents */
 BEGIN                          /* Virtual op implementation */
  @contents.putBody(Stio.Edit(@contents.getBody()));
  putLastUpdate();
  putAuthor(author);
 END Edit;

 OPERATION Send(mailboxes : List(Ref(Mailbox)); To:List(Text));
 VARIABLE                /* Send letter to listed recipients */
  mbox : Ref(Mailbox);
 BEGIN
  @DateSent:=Date.Today();
  @contents.putHeader(getAuthor(), To, @DateSent);
  FOREACH mbox IN mailboxes DO
   mbox$DeliverMail(SELF);
 END Send;
 .
 .
END Letter;
```

All public and private operations declared in the specification must be defined in the implementation. The implementation of an operation simply adds to the operation heading optional local variables and a compound statement. In addition, operations local to the implementation may be defined. Such operations are *only* available internally in the implementation for operating on SELF. SELF is a pseudo-variable designating the object instance that executes the code. In the example above SELF would implicitly be of type *Ref(Letter)*.

The representation of an object is defined in terms of *instance variables*. Instance variables in packettypes are called *attributes*. *Letter*, above, has two attributes, *contents* and *DateSent*. In statements '@' must precede an attribute. This is to remind programmers of their semantics (persistence), and that there is generally a difference in access cost between attributes and other variables. Instance variables of an object are not directly accessible from other objects. Hence, in order to let users modify the contents of a letter an *Edit* operation has been defined. *Edit* is declared in the specification of *Document* as a *virtual* operation. At run-time the object manager will dynamically bind an invocation on a virtual operation to the implementation provided by the most specialized type that the packet (or value) is an instance of. Declaring an operation virtual in a specification implies two things for implementors of subtypes. First, the operation may not be overridden in the specification of a subtype. Second, a subtype must have its own implementation for the operation (and it must conform to the inherited specification).

Virtual is used when the representation of an objecttype, which strongly affects the implementation of an operation, is not known at the level of generalization where the operation is specified. For example, the representation of *Letter* (i.e., contents : Message) is not known when the operations on *Document* are defined. Note that strong type checking is retained for both static and dynamic binding.[11]

[11] It is a sometimes held misconception that strong typing and dynamic binding are in conflict. Strong typing may be used

3.4. Object Management

In PAL, object instances are manipulated through expressions and statements. Below is an incomplete syntax for expressions:

```
Expression =
  [ Object InvokeOperator ] OperationCall
  { InvokeOperator OperationCall }.
```

Normally, an operation call (operation name and actual parameters) is preceded by an object (variable, literal, or the result of an expression) and an invoke operator (dot or dollar), e.g.:

```
mbox$DeliverMail(SELF);
```

The statement above means: invoke the *DeliverMail* operation on the packet referenced by the variable *mbox* and pass the value of *SELF* as parameter. The '$' operator is used for packettype operations and '.' for datatype operations. This reminds programmers about the difference in semantics and cost between operations on packets and datatype values.[12] If an operation is not preceded by an object, the object is assumed to be SELF. Operations can also be chained by directly invoking an operation on the result from another invocation. Chained operations are evaluated left-to-right.

Variables (and attributes, which are simply a special kind of variable) are distinct from datatype values in PAL. A variable should be seen as a container for a datatype value. When a datatype value is assigned to a variable, the previous datatype value will be replaced and garbage collected. Hence, assignment is not an operation on a datatype value, but on a variable. When an operation does not access any instance variables it is possible to invoke the operation directly on the type object, e.g.:

```
@DateSent := Date.Today();
```

Date is a primitive datatype and the operation *Today* is an operation independent of any particular instance. PAL does not have updatable class variables in the sense of Smalltalk-80,[13] i.e., variables accessible to all instances of a class. Only class constants are allowed. Class variables do not agree well with decentralization.

In order to instantiate a packettype the *Create* operation is used.

```
MyLetter := Letter$Create();
```

The *Create* operation does not access any instance variables and so may be invoked directly on the type. It creates a new instance of *Letter*, and calls an operation *Initialize* on the newly created instance, which initializes its attributes. Finally, the reference to the instance is returned. The reference must be saved, in a variable or attribute, or else the packet will become unreachable. Datatype values are created and initialized automatically by declaring a variable on the datatype.

Traditional programming languages provide an algebra (operators + operands) for manipulating data. In such languages the interpretation of an operator is fixed within a global and static type system. In PAL, and other OOPLs, the interpretation of an operation depends on the object on which it is invoked. For reasons of convenience PAL provides an "algebra" as an alternate syntax. However, the semantics is still object-oriented as the "algebra" is transformed into object-oriented form by the compiler. Hence, expressions such as 5+6 are transformed to 5.add(6).[14]

In addition to assignment statements and expressions there are also control structure statements, e.g., **if**, **while**, **repeat**, **foreach**, and **return**.

Compiled PAL is strongly typed. Sometimes, however, the type check must be deferred until run-time. For example, if the variable *doc* is declared to be of type *Ref(Document)*, then

```
doc$getDateSent()
```

would not be allowed since *getDateSent()* is only defined for letters. By requalifying *doc* to be of type *Letter* the expression becomes type correct:

```
doc:Ref(Letter)$getDateSent()
```

But, whether or not *doc* actually references a letter will only be known at run-time. The compiler will generate the code needed to perform a run-time type check.

The example above is called *downward* type qualification (downward in the inheritance graph). *Upward* type qualification is always possible. No run-time type checking is needed in this case.

3.5. Comments on Data Abstraction

In PAL property inheritance means strict inheritance of specification and default inheritance of implementation (as in Trellis/Owl [Schaf86]). It is possible to override operations declared in a supertype. For virtual operations the implementation in the supertype must be over-

for at least two purposes: safety and efficiency. It is possible to use strong typing for the purpose of safety alone and still have dynamic binding.

[12] Actually there is an additional reason for this. In order to be able to operate on a value of the *Ref* datatype and not on the packet it references there must be a way to syntactically distinguish the two cases.

[13] There are no global or pool variables either.

[14] Although the set of operators is currently fixed and hard-wired into the compiler, the operands do not have to be of a primitive type.

ridden in all subtypes. When overriding the implementation of virtual operations, the types of formal parameters may be generalized and formal results may be specialized. For standard operations overriding is only possible by overriding both the specification and the implementation. However, this overriding will not be effective on variables or references declared to the supertype, because of static binding.

In PAL it is possible to control the binding caused by virtual by using the invoke operator '$$' instead of '$'. This operator will cause static binding to be used, even if the operation is declared virtual. This is normally used only in the implementation of a virtual operation where access to the supertype's implementation is needed. For example the implementation of the operation *toListOfText* in *Letter* could invoke the implementation of this operation in *Document* by the code:

```
lst:List(Text);
.
.
lst := SELF:Ref(Document)$$toListOfText();
```

The '$$' operator is a more general mechanism than the use of the pseudo-variable 'super' in Smalltalk-80, since it may be used anywhere. It is less general than the mechanism in VBASE [Andre87] for method combination. However, the use of the '$$' operator for other purposes than invoking implementations of virtual operations in supertypes is discouraged, since it subverts the normal use of dynamic binding.

One problem with the virtual construct is that it is not possible for a designer of a subtype to alter the choice of which operations have been declared virtual in the supertype. In other words, it is not possible for the designer of a subtype to choose which inherited operations are to be invoked with dynamic binding of implementation. A designer could avoid the problem by always declaring every operation in the specification as virtual. As in Smalltalk-80, dynamic binding would then always be used. On the other hand this would also generate the additional overhead associated with dynamic binding, for every invocation.

In AVANCE we have added a mechanism to the object manager, and language constructs in PAL, which allows a designer to declare a supertype as *abstract* in the context section of a new subtype. This will cause the subtype to inherit only the specification of the supertype. Invocations of any operation on instances of the subtype will be dynamically bound, even if the invocation is made from a type compiled against the supertype. We use one bit of a bit-vector component in packet references (see figure 5 in section 4.1) to indicate whether or not dynamic binding always should be used. Instances created from types which have declared one or more supertypes as abstract will get this bit set in the reference returned by the *Create* operation.[15] Normal invocations, which use static binding, will have the added overhead of a test on one bit which is negligible. Invocations on virtual operations will not be affected. Invocations which would have been static, but which find the bit set will have an overhead which is greater than that of virtual. This is because the object manager has to convert the static binding invocation into one of dynamic binding and this will involve extra searching at run-time.

Since declaring a supertype as *abstract* will cause extra overhead on all invocations on all instances of the new type (and all instances of subtypes to the new type), the designer might be reluctant to use this mechanism. Still, the mechanism is there for use in case it is needed.

In PAL encapsulation is also upheld between a type and its subtypes. Thus, it is not possible to directly access instance variables declared in a type from one of its subtypes. That is why operations such as *putLastUpdate* and *putAuthor* are invoked from the *Edit* operation in *Letter* in order to modify attributes in *Document* (of which *Letter* is a subtype). This strong form of encapsulation is important from a software engineering perspective, as argued by Snyder [Snyde87]. For packettypes it also naturally maps to the mutual independence of packet-slices at the object manager level (see section 4). Wegner has argued that distribution is inconsistent with inheritance [Wegne87]. AVANCE/PAL is a counterexample to this. The different slices of one packet may be distributed over several nodes. This is possible because the representation of a packettype is not inherited explicitly. Communication between the different types of one packet is only allowed through packet operations (having the indirection of a remote procedure call) and not by shared instance variables.

For datatype values the argument for this strict form of encapsulation is not quite as strong. The representation of a datatype value is always maintained as one unit and can not be distributed in the way a packet can be. In fact it has been argued that access to the representation of a type from descendants is often desirable for reasons of reusability ([Meyer88] pp 272-274). Because of this we may in the future relax the encapsulation of datatypes (not packettypes) to allow subtypes to access the instance variables of a supertype.

[15] Datatype values are not identified and manipulated by references, but each value of a user-defined datatype internally has a reference to its executable type slice. The corresponding bit may then be set in this reference.

3.6. Local Consistency and Interpreted PAL

Since PAL is strongly typed and at the same time type extensible without going outside the system, there must be some "loopholes" for adding new functionality to the system. One such loophole is the virtual construct which through dynamic binding allows a type to operate on instances of yet to be defined types. This requires that specifications be known a'priori at least for entry-points to applications. A designer may create a subtype to packettype *Application* which has a virtual operation *main* declared in its specification. This gives a standardized entry-point and for most purposes this is enough.

There are still some cases where it may be needed to construct calls on objects where the objects type specification is unknown (or not completely known) at compile-time. PAL has a pre-defined datatype *Invocation* for this purpose:

```
DATATYPE Invocation;
  SPECIFICATION
  PUBLIC
    OPERATION putInstance(instance : Value) : Invocation;
    OPERATION putOpName(opname : Text) : Invocation;
    OPERATION putArgs(args : List(Value)) : Invocation;
    OPERATION putRetVals(retvals : Integer) : Invocation;
    OPERATION invoke() : Invocation;
    OPERATION getRetVal(retval : Integer) : Value;
END Invocation;
```

The following would prepare a call on the *Edit* operation of a document:

```
invoc : Invocation;
doc : Ref(Document);
lst : List(Value);
author : Text;
 .
 .
invoc.putInstance(doc).putOpName("Edit").
putArgs(lst.insert(author));
```

A value of *Invocation* is just like any other datatype value, it may be passed around as a parameter, kept persistently in an attribute, etc. To actually evaluate the invocation the operation *invoke* is used. *PutRetVals* and *getRetVal* are used if return values are of interest. The arguments passed to an operation when using an *Invocation* value consists of a list of values. A run-time check (downwards qualification) will be done for each argument. Similarly, return values are simply values and must be qualified to something more specific if they are to be used. The actual invocation is done using the same mechanism (in the object manager) that is used for the virtual construct.

We require objects to maintain *local consistency*. By this we mean that an object may operate on other objects of any type as long as it has certified by qualification that the other object conforms in specification to one of the types in its own context. In the case of parameter passing it is up to the invoker to supply type correct arguments and the invokee to suply type correct return values.

For purposes of bootstrapping and prototyping AVANCE has a second interpreter, besides the p-code interpreter (see fig 1.). This interpreter directly executes a subset of PAL. In operating systems terms it corresponds to a command language shell. Input to the PAL interpreter is a List(Text) object where it parses one line at a time and attempts to execute it. It uses a less strong form of typing than compiled PAL and is prepared to handle any of a number of exceptions generated by the object manager layer. All invocations, except invocations on primitive datatypes, use the dynamic binding mechanism of the object manager. If the interpreter encounters the operation *Interact()* it will obtain lines of input interactively from the user until the user writes the command *return()*.

3.7. Homogeneity

The distinction between packettypes and datatypes in PAL is a disadvantage from the perspective of homogeneity. It might be argued that the decision on whether an object should be persistent or not should be made automatically by the system. We introduced the distinction mainly because it simplified the implementation of AVANCE enormously. For example, the "small object problem" is considerably reduced. A programmer which is unsure whether to implement an objecttype as a datatype or as a packettype should probably implement it as a datatype. A corresponding packettype with its increased overhead is easily made if needed, by encapsulating a datatype value in an attribute and writing the corresponding (trivial) packet operations. The reverse is also possible: encapsulating a reference to a packet in a datatype. But this will not undo the overhead.

Not only programs and data, but also meta-data (objecttypes) should be objects in an OMS. An objecttype is a description of a set of instances in the system. However, objecttypes can also be regarded as instances of a *meta objecttype*. To represent objecttypes PAL has a predefined packettype *TypeDef*. *TypeDef* is specialized into *DatatypeDef* and *PackettypeDef*. This view is necessary since we want to instantiate new types from *within* the system and add them incrementally without having to recompile the whole system. A new packettype (or datatype) is created by making an instance of *PackettypeDef* (or *DatatypeDef*). For example, *Letter* is instantiated from *PackettypeDef* and made a subtype of

Document, which is also instantiated from *PackettypeDef* but a subtype of *Packet*. *Letter* may in its turn be instantiated into ordinary letters. The meta objecttypes define a representation for PAL source code, p-code (compiled code), and operations for manipulating the representation, e.g., *EditSpecification*, *EditImplementation*, *CompileSpecification*, and *CompileImplementation*.

In order to take advantage of the uniformity a "data-dictionary" facility is created by the compiler. A few extra operations are generated automatically for every objecttype.[16] Among these are *getTypename*, *getOperations*, and *getParameters*. Hence, any object can be queried about its type properties. This is good both for learning about objecttypes (in interpreted PAL) and for constructing objects where the logic is affected by "meta-data".

3.8. Processes

An AVANCE process is the root of an invocation tree. Processes are created and manipulated using the parameterized *Process* datatype which is similar to the *CBox* objects of ConcurrentSmalltalk [Yokot87] and the *futures* concept of actors [Agha87]. Any PAL expression (not statement) may be executed in its own new process by placing it in square brackets. The following code segment starts separate processes for sending a letter to all mailboxes:

```
FOREACH mbox IN mailboxes DO
  [ mbox$DeliverMail(SELF) ];
```

The value of the bracketed expression above is of type *process()*.[17] In the above case the invoker is not interested in any result returned by the expression. To be able to obtain the result of executing a process the process value must be retained.

```
prc : Process(Integer);
itg : Integer;
 .
 .
prc := [1+2];
 .
 .
itg := prc.result();
```

In this example the integers 1 and 2 are added in their own process. Later the result is fetched and assigned to the variable itg. The expression [1+2] returns immediately with a process value which is assigned to prc. The prc.result() expression synchronizes with the other process to obtain the result, i.e., if the other process is finished then the value is returned, otherwise the calling process will wait.

The only means of communication between processes is through shared access to packets and the initial and final passing of arguments and return values via the *Process* object.

4. The AVANCE Virtual Machine

The distinction between packets and datatype values is reflected at all three abstraction levels of the architecture (see fig 1). A packet at the level of PAL is represented by a set of packet-slices at the level of the object manager. The object manager provides a single level store for packet-slices, handles references to packet-slices, and controls the execution of processes. What unites conceptually a set of packet-slices into one abstract packet is a common identifier. A packet-slice consists of a set of persistent attributes belonging to one specific packettype. The usual reason that a packet is represented by more than one slice is property inheritance. For each type that a packet is an instance of there will correspond a slice, containing exactly the attributes belonging to that specific type. However, the slices of a packet are independent from each other from the perspective of the object manager, and it does not place any restrictions on which slices of a packet are created and destroyed. Restrictions, such as those implied by property inheritance, are defined and enforced at the PAL level. Inheritance is realized at the object manager level by a flexible invocation mechanism which allows type slices to delegate [Stein87] the implementation of operations to other type slices. A packet could in principle dynamically change its type structure or even consist of types which are unconnected by inheritance. A packet could also be distributed over a network, by having slices at different locations. Datatype values on the other hand always have their whole representation kept together as one unit by the object manager.

At the object manager level, a crude form of object orientation is provided by an invocation/return cycle operating with packet-slices and references to slices. Datatype values are simply datastructures at this level. Within the context of a compiled operation, an interpreter is given control. The interpreter performs the actual execution of operations on both packet-slices and datatype values. When the operation terminates, control is returned to the object manager. The passing of call and return parameters for operations of non-primitive types is also handled by the object manager. Since parameters are passed by value they will be

[16] They are virtual operations in both *Packet* and *Value*.

[17] Because the *DeliverMail* operation of *Mailbox* does not return anything the *Process* value is parameterized with nothing.

copied, but the copying is deferred until the parameter is actually accessed by the interpreter.

All the primitive datatypes use a low level representation which the object manager understands enough to preserve structure and content. The object manager takes care of transporting such values between secondary storage and the cache, and between the cache and the interpreter(s). It is possible to add new primitive datatypes, but this requires modification to the interpreter layer and a static relinking of the system. This is expected to be relatively rare. It would only be done for datatypes which need to be optimized, or to add basic functionality missing from the current set of primitives.

The normal way to add a new object type to the system is to define it in PAL, compile it and install it. The installation of a new packettype or datatype is effectively a form of dynamic linking which does not disrupt the normal operation of the system.

4.1. References and Invocation

Figure 5 illustrates a packet reference as it is represented in the object manager.

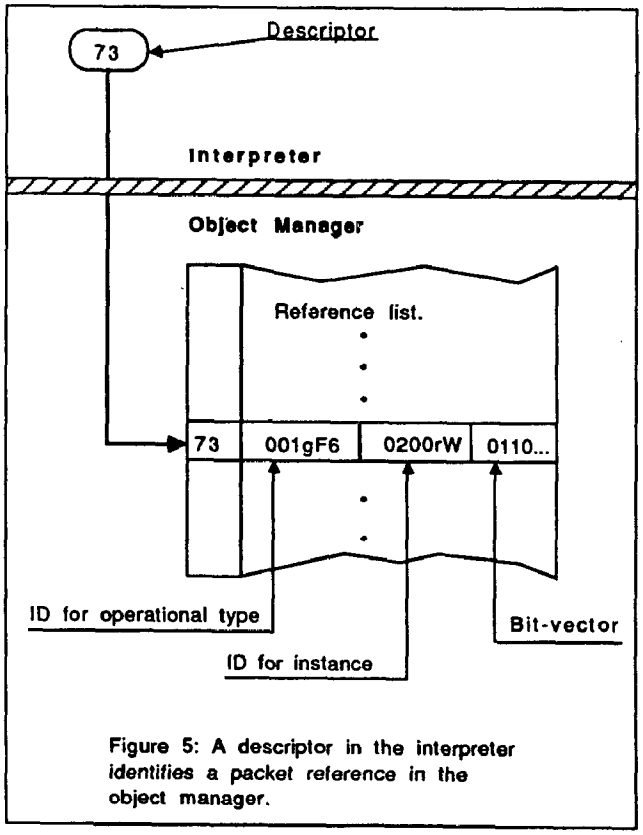

Figure 5: A descriptor in the interpreter identifies a packet reference in the object manager.

It contains two identifiers plus a bit-vector used by the object manager to maintain information about the reference. The two identifiers in combination identify a packet-slice. Packet references never leave the object manager. The interpreter layer manipulates references indirectly by using descriptors. For each packet-slice the object manager maintains an associated list of references "owned" by that slice. A descriptor identifies a reference in the list. This ensures that the basic capability based protection mechanism [Levy84] of the object manager is not violated. When a descriptor is part of a datatype value which is assigned to a persistent attribute, the object manager will make the corresponding reference persistent.

The object manager handles the invocation of operations on packet-slices. In other words, when given a descriptor, and an operation to invoke, it finds the correct packet-slice of both the type (holding the code) and the instance (holding the instance variables) and starts the correct interpreter to execute the code of the operation. To find the type slice the object manager creates a new reference based on the reference identifying the instance. The operational type identifier of the instance reference becomes the instance identifier of the new reference. The operational type identifier of the new reference is an identifier for *executables* known to the object manager.

4.1.1. Transaction Management

The object manager employs a low level version management mechanism inspired by the work of Reed [Reed78]. The details of this mechanism are explained in another paper [Björn88], only an outline is given here. Packet slices are maintained in time-stamped versions by the object manager. A packet operation which updates a slice will generate a new version of that slice. Versions are tentative until the operation which generates them has committed. Old versions of a slice are kept by the object manager for at least the time needed to complete the transaction generating a new version. Versions may also be kept for longer if applications need them. The unit of synchronization and system version management is the packet slice. The unit of action on a packet-slice is the packet operation. A nested transaction scheme is used where each packet operation is regarded as a (sub)transaction. A top (independent) transaction is indicated by using '#' as invoke operator on a packet operation. When an operation is invoked the object manager prepares for access to three different slice versions in the cache.

- The *operational type version* slice (always located, read-only).
- The *instance read version* slice (may be omitted, read-only).
- The *instance write version* slice (may be omitted, both read and write allowed).

The code for all specific (non-inherited) operations of one type resides in a slice belonging to the type. The

object manager must locate the correct version of this slice if there are several versions due to the type having been updated. The read version is the "latest" version of the instance slice valid before the current invocation. The write version is the new slice version to be generated by the current invocation.

Since the read version and the write version may be omitted there are four kinds of invocations:

- *Read-only.* The operation accesses attributes, but does not assign to them or apply modifying datatype operations to their values. The write version is omitted.
- *Modifying.* The operation both reads and writes attributes and makes some reads before writes. Both the read and write version is needed.
- *Independent.* The operation only writes (assigns) to attributes, or assigns to attributes before reading them. The read version is omitted.
- *No-instance.* The operation does not access attributes. Both the read and write version is omitted.

This classification of operations is used by the object manager to reduce overhead and to increase potential concurrency. The classification is done automatically by the PAL compiler. For example, the no-instance class does not need any synchronization. It is similar to what Hewitt has called an *unserialized actor* [Hewit84], except that here it is not the packet (or actor) as a whole being serializable or not, but the operation (or part of the behavior). Such an operation does not access the instance as such. The operations of datatypes defined in PAL are much like no-instance packet operations from the object managers point of view. The object manager will only need to fetch the operational type version slice since for a datatype value there is no instance slice. The value instead resides on the local stack.

4.2. Decentralization

Although functions for handling more than one node have not been fully implemented yet in AVANCE we think it important to describe how the AVANCE architecture is appropriate for decentralization.

4.2.1. Object Identity

Object identity has been recognized as important [Khosh86] particularly in a system such as AVANCE with both persistent objects and decentralization. The distinction between packets and datatype values significantly reduces the number of identifiers needed. For AVANCE we consider the advantage of a common identification scheme important enough to sacrifice some of the symmetry of the logical network (see figure 2). Each node has a sequence of unused identifiers. When a new packet is created at a node, an identifier is taken from the sequence and associated with the new packet. When a node is about to run out of identifiers, it communicates with one specific other node, which has been designated as the *identifier generator node,* (IGN) and replenishes the sequence. This introduces a slight centralization in the system, since all nodes will be dependent on the IGN. The event of running out of identifiers can however be made as infrequent as desired by making the replenishing sequence sufficiently large. The margin allowed before ordering new identifiers from the IGN may also be made large to give ample time for the order to be effected. Furthermore, if the IGN intends to close or otherwise become unsuitable as the IGN, it may dynamically request another node to take over the IGN function. This is done by moving a special IGN-packet to the new IGN. If the IGN node does close, or become unavailable to some nodes, then these will be forced to close when they run out of identifiers.

There are several other details and enhancements to this scheme not covered here,[18] the main point here being that identifier generation is not a big problem.

4.2.2. Low Level Representation

For the representation of values of the primitive datatypes we have designed a special language for defining composite data structures. We have implemented functions in the object manager, that given such a data structure description, can generate a primary storage structure, and a corresponding serialized structure for use on secondary storage and for node-local communication. The normal AVANCE user/programmer does not have to be concerned with these low level data structures. They are only visible to the implementor of a primitive datatype. User defined packet/datatypes are defined as abstract datatypes and use the primitive datatypes (as abstract datatypes) for defining their internal representation.

4.2.3. Communication Protocol

Because the invocation of packet operations has a remote procedure call semantics, it allows for any of a number of lower level communication protocols, for example the Sun RPC/XDR facility [Sun86] is one possibility. The representation form described in the previous section is not machine independent. For data to be communicated between nodes we will not use our own

[18] One could have a hierarchy of IGN's to reduce the problem of one IGN becoming a bottleneck. This could improve both performance and reliability.

serialization routines, but instead serialize to a representation which is machine independent, such as XDR. On top of this we need our own protocol for movement of packet(slice)s between nodes, remote packet (slice) operation invocation, and transaction management with two phase commit [Gray78]. The transaction management protocol will not be further discussed.

Movement of packets

A packet slice can only reside at one node at a time, but replicas of the versions generated at the node may be given to other nodes. The meaning of a slice residing at a node is that the node has the *right* to update the slice, i.e., generate new versions of the slice localy by executing modifying or independent operations localy. Other nodes may only acquire replicas of already existing slice versions. To move a packet slice from one node to another, first the latest valid slice is replicated on the destination node, then the update right is moved. This has to be done as an atomic operation and uses the same transaction protocol as used with normal packet operations.

Remote invocation

The invocation of a packet operation will result in the object manager trying to locate the correct slice of both the type and the instance involved from the information in the reference. If the correct type slice is not found locally then the invocation will fail. If the type slice is found, then the object manager attempts to look up the slice of the instance belonging to that type. If this fails then the object manager will "guess" on which node the instance slice is located and pass the original invocation on to that node. If the other node succeeds it simply returns the result of the invocation. If it fails for the same reason that the original node failed, the other node will make a "guess" on the location of the instance. However, instead of directly passing the invocation on to its guess the other node returns the guess to the original node which makes a new guess, perhaps based on the guess returned by the previous try etc. In the worst case the original node will attempt to communicate with every other node it knows of. Once the location of the instance has been found the object manager will remember this location as a first guess for the next invocation on the same instance slice. The idea of guessing or using a "soft" form of addressing for locating objects at other nodes has been borrowed from the Xerox Clearinghouse [Oppen83].

An invocation of an operation which is read-only can sometimes be executed locally even if the instance slice does not reside at the node. This will be possible if the node has old versions (or replicas) of the slice and the reference used in the invocation is bound to a time where an old version is known to be valid.

4.2.4. Distributing a new type

The pre-defined types (primitive datatypes and system defined packettypes) form a substrate common to all nodes. When a new type is defined in the system it will be represented as a packet and thus have its own global identifier. It is not enough to just compile a type to make it available for use as a type. The compiled type must be installed. This is done on a per-node basis, usually starting with the node on which the type was created. The installation of a type at a node creates an executable *operational type* slice of the type at the node. Thus, a type maintains its identity over all nodes where it is installed. The installation of a type involves issues of type version management discussed in [Björn88].

5. Summary

AVANCE is an OMS integrating programming and database management by providing a single level store. The object space is partitioned into *packets* and *datatype values*. This simplifies the implementation of the system and also gives the PAL programmer control over which objects should be persistent or shared. The AVANCE architecture provides three levels of abstraction: the low-level object manager, the pseudo-code interpreter, and the PAL programming language. The architecture is also geared towards decentralization, by the remote procedure call semantics of packet operations, by the very large identifier space, and by the use of immutable versions of slices of packets generated by atomic actions.

The object manager provides a low level object oriented and capability based operating system, including an invocation mechanism that synchronizes processes by making operations on packet-slices atomic. The invocation of an operation on a packet (slice) is the point of indirection where dynamic binding may be applied to procedure implementation, instance version, type version and instance location. The possibility of adding new interpreters and compilers allows for a multiparadigm language environment. Although the basic paradigm provided by the object manager is object oriented, the language used for writing the implementation of a type's operations could be any language for which a compiler and/or interpreter is provided.

6. Acknowledgements

Besides the authors, the following people have been involved in the specification and implementation of AVANCE: Matts Ahlsén, Stefan Paulsson and Dr. Christer Hultén.

References

[Agha87].
G. Agha and C. Hewitt, "Actors: A Conceptual Foundation for Concurrent Object-Oriented Programming," in *Research Directions in Object-Oriented Programming*, ed. B. Shriver and P. Wegner, pp. 49-74, The MIT Press, 1987.

[Ahlse84].
M. Ahlsen, A. Björnerstedt, S. Britts, C. Hulten, and L. Söderlund, "An Architecture for Object Management in OIS," *ACM Transactions on Office Information Systems*, vol. 2, no. 3, July 1984.

[Ahlse85].
M. Ahlsen, A. Björnerstedt, and C. Hulten, "OPAL: An Object-Based System for Application Development," *IEEE Database Engineering Bulletin*, vol. 8, no. 4, pp. 31-40, Dec. 1985.

[Ahlse87].
M. Ahlsen, A. Björnerstedt, S. Britts, and S. Paulsson, "PAL Reference Manual," WP No 125, SYSLAB, University of Stockholm, Stockholm, Sweden, 1987.

[Andre87].
T. Andrews and C. Harris, "Combining Language and Database Advances in an Object-Oriented Development Environment," *Object-Oriented Programming Systems, Languages and Applications*, pp. 430-440, Oct. 1987.

[Birtw73].
G. Birtwistle, O-J Dahl, B Myhrhaug, and K. Nygaard, *Simula Begin*, Auerbach, Philadelphia, 1973.

[Björn88].
A. Björnerstedt and C. Hulten, "Version Control in an Object-Oriented Architecture," in *Object-Oriented Concepts, Applications, and Databases*, ed. W. Kim and F. Lochovsky, Addison-Wesley, 1988.

[Black86].
A. Black, N. Hutchinson, E. Jul, and H. Levy, "Object Structure in the Emerald System," *Object-Oriented Programming Systems, Languages and Applications*, pp. 78-86, Portland, Oregon, 1986.

[Carey86].
M.J. Carey, D.J. DeWitt, D. Frank, G. Graefe, M. Muralikrishna, J.E. Richardson, and E.J. Shekita, "The Architecture of the EXODUS Extensible DBMS," *Proceedings: 1986 International Workshop on Object-Oriented Database Systems*, pp. 52-65, Pacific Grove, California, 1986.

[Cox86].
B.J. Cox, *Object Oriented Programming An Evolutionary Approach*, Addison-Wesley Publishing Company, 1986.

[Goldb83].
A. Goldberg and D. Robson, *Smalltalk-80 The Language and its Implementation*, Addison Wesley, 1983.

[Gray78].
J.N. Gray, "Notes on database operating systems," in *Operating Systems*, ed. R. Bayer, R.M. Graham, and G. Seegmuller, pp. 393-481, Springer-Verlag, Berlin, 1978.

[Gray86].
J.N. Gray, "An Approach to Decentralized Computer Systems," *IEEE Transactions on Software Engineering*, vol. SE-12, no. 6, pp. 684-692, 1986.

[Hewit84].
C. Hewitt and P. deJong, "Open Systems," in *On Conceptual Modelling*, ed. M.L. Brodie, J. Mylopoulos and J.W. Schmitt, pp. 147 - 164, Springer-Verlag, New York, 1984.

[Hoare74].
C.A.R. Hoare, "Monitors: an operating system structuring concept," *Communications of the ACM*, vol. 17, no. 10, pp. 549-557, Oct. 1974.

[Horni87].
M.F. Hornick and S.B. Zdonik, "A Shared, Segmented Memeory System for an Object-Oriented Database," *ACM Transactions on Office Information Systems*, vol. 5, no. 1, pp. 70-95, Jan. 1987.

[IEEE85].
IEEE, "Special Issue on Object-Oriented Systems," *IEEE Database Engineering Bulletin*, vol. 8, no. 8, Dec. 1985.

[IEEE86].
IEEE, *Proceedings: 1986 International Workshop on Object-Oriented Database Systems*, IEEE, Sep. 1986.

[Khosh86].
S.N. Khoshafian and G.P. Copeland, "Object Identity," *Object-Oriented Programming Systems, Languages and Applications*, pp. 406-416, Portland, Oregon, Sep. 1986.

[Levy84].
H.M. Levy, *Capability - Based Computer Systems*, Digital Press, 1984. ISBN 0-932376-22-3

[Lisko81].
B. Liskov, R. Atkinson, T. Bloom, E. Moss, J.C. Schaffert, R. Scheifler, and A. Snyder, *CLU*

Reference Manual, Springer-Verlag, Berlin, Heidelberg, 1981.

[Lisko83].
B. Liskov and R. Scheifler, "Guardians and Actions: Linguistic Support for Robust, Distributed Programs," *ACM Transactions on Programming Languages and Systems*, vol. 5, no. 3, pp. 381-404, July 1983.

[Maier86].
D. Maier, J. Stein, A. Otis, and A. Purdy, "Development of an Object-Oriented DBMS," Tech. Rep. CS/E-86-005, Oregon Graduate Center, April 1986.

[Meyer88].
B. Meyer, *Object-oriented Software Construction,* Prentice Hall, 1988.

[Niers87].
O.M. Nierstrasz, "Active Objects in Hybrid," *Object-Oriented Programming Systems, Languages and Applications*, Orlando, Florida, Oct. 1987.

[Oppen83].
D.C Oppen and Y.K. Dalal, "The clearinghouse: A decentralized agent for locating named objects in a distributed environment," *ACM Transactions on Office Information Systems*, vol. 1, no. 3, July 1983.

[Purdy87].
A. Purdy, B. Schuchardt, and D. Maier, "Integrating an Object Server with Other Worlds," *ACM Transactions on Office Information Systems*, vol. 5, no. 1, pp. 27-47, Jan. 1987.

[Reed78].
D.P. Reed, "Naming and Synchronization in a Decentralized Computer System," MIT/LCS/TR-205, MIT Laboratory for Computer Science, Cambridge, Massachusetts, Sep. 1978.

[Schaf86].
C. Schaffert, T. Cooper, B. Bullis, M. Kilian, and C. Wilpolt, "An introduction to Trellis/Owl," *Object-Oriented Programming Systems, Languages and Applications*, pp. 9-16, Portland, Oregon, Sep. 1986.

[Snyde87].
A. Snyder, "Inheritance and the Development of Encapsulated Software components," in *Research Directions in Object-Oriented Programming*, ed. B. Shriver and P. Wegner, pp. 165-188, MIT Press, Cambridge, Massachusetts, 1987.

[Stein87].
L.A. Stein, "Delegation Is Inheritance," *Object-Oriented Programming Systems, Languages and Applications*, pp. 138-146, Orlando, Florida, Oct. 1987.

[Stone81].
M. Stonebraker, "Operating System Support for Database Management," *Communications of the ACM*, vol. 24, no. 7, July 1981.

[Stone86].
M. Stonebraker and L.A. Rowe, "The design of Postgress," *ACM SIGMOD Proceedings*, pp. 340-355, Washington DC, June 1986.

[Strou86].
B. Stroustrup, *The C++ Programming Language,* Addison-Wesley, 1986.

[Sun86].
Sun, "Remote Procedure Call Programming Guide," Revision B, Sun Microsystems, 17 Feb. 1986.

[Svobo84].
L. Svobodova, "Resilient Distributed Computing," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 3, pp. 257-268, May 1984.

[Wegne87].
P. Wegner, "Dimensions of Object-Based Design," *Object-Oriented Programming Systems, Languages and Applications*, pp. 168-182, Orlando, Florida, Oct. 1987.

[Yokot87].
Y. Yokote and M. Tokoro, "Experience and Evolution of ConcurrentSmalltalk," *Object-Oriented Programming Systems, Languages and Applications*, pp. 406-415, Oct. 1987.